Python datetime: trabajando con fechas

Yan Orestes
29/09/2020

Tomemos un ejemplo concreto de **fechas y horas** para poder programar en Python y con el módulo `datetime`: una compañía nos contrató para implementar su sistema de puntos, controlando cuándo llega y se va un empleado. El sistema debe mostrar la fecha y la hora en cada registro, como confirmación para el empleado.

date de datetime

Conocemos el módulo datetime de la biblioteca nativa de **Python**, por lo que incluso sabemos cómo obtener la fecha actual a través de la clase date, simplemente impórtelo y llame al método **`today ()`**:

```
from datetime import date

fecha_actual = date.today()
print(fecha_actual)
```

Como esperado:

```
2020-9-1
```

Pero calma. ¡Sería bueno si pudiéramos imprimir la fecha en formato **DD / MM / AAAA** para evitar confusiones! El problema es que el **`date`** fuerza automáticamente el estándar ANSI

cada vez que intentamos imprimir.

Formateando nuestra fecha en un string.

Dado que la clase `date` puede proporcionar cada sección de la fecha por separado, podemos resolver este problema con un **formato de string**:

```
fecha_en_texto = '{}/{}/{}'.format(fecha_actual.day, fecha_actual.month, fecha
```

Lo que resulta en:

```
1/9/2020
```

Mejor, pero todavía no es exactamente lo que queríamos. Tenga en cuenta que tanto el día como el mes no tienen el prefijo **0**, lo que no está en el estándar deseado. En este caso, incluso podríamos evitar esto simplemente agregando un **0** antes:

```
fecha_en_texto = '0{}/0{}/{}'.format(fecha_actual.day, fecha_actual.month,fech
```

Lo que funcionaría, pero causaría problemas si el día o mes fuera mayor o igual a 10:

```
010/010/2020
```

Formatear fechas en strings usando el método strftime ()

Para evitar más complicaciones, la clase de fecha resuelve esto con un único método: **`strftime ()`**, que toma como parámetro el formato que queremos en nuestra string de fecha, por lo tanto, nos da más libertad para decidir cómo queremos mostrar la fecha.

Este formato utiliza códigos mejor explicados en la documentación. Al final del texto, también damos una breve explicación sobre ellos. En nuestro caso se ve así:

```
fecha_en_texto = fecha_actual.strftime('%d/%m/%Y')
```

```
print(fecha_en_texto)
```

E asi:

```
01/09/2020
```

En el caso de nuestro ejemplo que resultó en 010/010/2020:

```
10/10/2020
```

Ahora solo necesitamos encontrar una manera de almacenar el tiempo también. ¿Quién puede encargarse de eso? Como habrás adivinado, el mismo módulo `datetime` desde el que importamos la clase `date` también tiene clases que facilitan la manipulación de los tiempos.

El tipo de fecha y hora para cuidar de fechas y horas juntos

Si bien podemos usar el tipo time, destinado exclusivamente para tiempos, el módulo nos brinda una solución mucho más apropiada para nuestro problema con el tipo datetime: **sí, tiene el mismo nombre que el módulo, ¡cuidado con la confusión!**

Una de las ventajas de la clase `datetime` es que puede manejar la fecha y la hora al mismo tiempo. La única diferencia en nuestro uso es que, en lugar del método `today ()`, usaremos el método **`now ()`**:

```
from datetime import datetime

fecha_y_hora_atuais = datetime.now()
fecha_y_hora_en_texto = fecha_y_hora_actuales.strftime('%d/%m/%Y')

print(fecha_y_hora_en_texto)
```

El resultado es como el anterior:

```
01/09/2020
```

Aunque ya estamos usando la clase `datetime`, que incorpora la hora, necesitamos declarar en el formato que pasamos como parámetro a `strftime ()` que también queremos mostrar la hora y los minutos:

```
fecha_y_hora_en_texto = fecha_y_hora_actuales.strftime('%d/%m/%Y %H:%M') print
```

Y ahora si:

```
01/09/2020 12:30
```

¡Perfecto! Hasta ahora hemos aprendido a tomar la fecha actual con `date`, `datetime` y incluso hemos aprendido a **formatear fechas**, transformándolas en strings. ¿Pero y si tuviéramos que ir para otro lado?

Convertir una strong a datetime

Si tuviéramos una string de date y quisiéramos transformarla en `datetime`, ¿qué haríamos? De nuevo, un método simple resuelve todo, esta vez **`strptime ()`**, desde la misma clase `datetime`:

```
from datetime import datetime

fecha_y_hora_em_texto = '01/09/2020 12:30'
fecha_y_hora = datetime.strptime(fecha_y_texto, '%d/%m/%Y %H:%M')
```

El problema de la zona horaria

Revisé la configuración de una de estas computadoras y descubrí que su reloj estaba en una zona horaria diferente a la de Bogotá, donde se encuentra la compañía.

No podemos dejar que el **tiempo en nuestro programa dependa de cada máquina**, porque no podemos garantizar que todas las máquinas que ejecutan este programa estén en la zona horaria que deseamos. Lo ideal, entonces, sería forzar la zona horaria para donde quieras.

Zona horaria con clase de timezone

A partir de **Python 3**, tenemos la clase timezone, también del módulo `datetime`:

```
from datetime import datetime, timezone

fecha_y_hora_actuales = datetime.now()
zona_horaria = timezone()
print(zona_horaria)
```

Veamos qué se imprime en la pantalla:

```
Traceback (most recent call last):
  File "teste.py", line 4, in >
    fuso_horario = timezone()
TypeError: Required argument 'offset' (pos 1) not found
```

La excepción **`TypeError`** que recibimos indica que no se encontró el argumento **`offset`**, esperado en el constructor timezone. Realmente no hicimos ese argumento. Pero, ¿qué significa?

El parámetro `offset` representa la diferencia entre la zona horaria que queremos crear y el tiempo universal coordinado (UTC). En nuestro caso, en Bogotá, tenemos una diferencia de -5 horas, mejor conocida como **UTC-5**. Sabiendo esto, intentemos nuevamente:

```
zona_horaria = timezone(-5)
print(zona_horaria)
```

Ahora que hemos configurado el parámetro de compensación, veamos qué aparece en la pantalla:

```
Traceback (most recent call last):
  File "teste.py", line 4, in <module>
    fuso_horario = timezone(-5)
TypeError: timezone() argument 1 must be datetime.timedelta, not int
```

Esta vez, el mensaje de excepción es diferente, lo que indica que el argumento pasado al constructor timezone debe ser del tipo **`datetime.timedelta`**, no un entero, que es lo que pasamos.

Calcular la diferencia horaria con la clase timedelta

La clase **timedelta** tiene el propósito de representar una duración y, en nuestro caso, una diferencia entre horarios. Ahora, ejemplifiquemos en una variable e intentemos imprimir, para ver qué sucede:

```
diferencia = timedelta()
print(diferencia)
```

Mira lo que apareció:

```
0:00:00
```

De acuerdo, 0 días, 0 horas y 0 minutos. Pero necesitamos que nuestro objeto `timedelta` se corresponda con la diferencia **UTC**, a -5 horas:

```
diferencia = timedelta(-5)
print(diferencia)
```

¿Qué aparece ahora?

```
-5 days, 0:00:00
```

¿Que? -¿5 días? ¡Queríamos -5 horas! El problema es que el constructor timedelta recibe varios otros argumentos más allá de la hora, en ese orden:

- ***days*** (dias)
- ***seconds*** (segundos)
- ***microseconds*** (microsegundos)
- ***milliseconds*** (milisegundos)
- ***minutes*** (minutos)
- ***hours*** (horas)
- ***weeks*** (semanas)

Entonces, si solo le enviamos un **-5**, ese número se interpreta como si fuera en días. Podemos pasar **0** para los primeros 5 parámetros y **-5** para las horas, pero esto es un poco extraño, teniendo en cuenta que, de hecho, **solo queremos definir las horas**.

Usando la funcionalidad de Python de **parámetros con nombre**, es posible especificar que estamos configurando el parámetro horas (*hours*), de la siguiente manera:

```
diferencia = timedelta(hours=-5)
print(diferencia)
```

Y ahora:

```
-1 day, 19:00:00
```

Bueno, si ponemos -5, ¿por qué apareció todo esto? Resulta que `timedelta` entiende **-5 horas** como **0 días, 0 horas y 0 minutos - 5 horas**, es decir, **-1 día, 19 horas**.

Resolviendo el problema de zona horaria

Ahora, creemos un objeto de zona horaria correspondiente a **UTC-5**, indicando esta diferencia con respecto a **UTC** como parámetro del constructor:

```
zona_horaria = timezone(diferencia)
print(zona_horaria)
```

Tenemos justo lo que queríamos:

```
UTC-05:00
```

Finalmente, podemos convertir el tiempo de la máquina al de Bogotá, utilizando el método **`astimezone ()`**:

```
fecha_y_hora_bogota = fecha_y_hora_actuales.astimezone(zona_horario)
fecha_y_hora_bogota_en_texto = fecha_y_hora_bogota.strftime('%d/%m/%Y %H:%M')

print(fecha_y_hora_bogota_en_texto)
```

Ahora todo está estandarizado:

```
01/09/2020 12:30
```

Ahora tenemos todo arreglado con la zona horaria, pero ¿y si le mostramos nuestro código a otro programador, él entendería lo que significa **-5**? ¿No es muy fácil, verdad?

De hecho, para cada zona horaria tendríamos que investigar cuál es su diferencia con UTC, que es un problema molesto. ¿No hay una manera más simple y elegante de resolver este problema?

Resolviendo el problema de zonas horarias con pytz

La comunidad de Python, ante esta necesidad, creó varias bibliotecas para facilitar la manipulación de *timezones*,, como pytz. Para instalar pytz, puede usar pip desde la terminal:

```
pip install pytz
```

Es probable que la instalación en sistemas basados en UNIX requiera permiso sudo.

Una vez instalado, podemos importar su clase `timezone` y es fácil obtener la zona horaria que queremos:

```
from datetime import datetime
from pytz import timezone

fecha_y_hora_actuales = datetime.now()
zona_horaria = timezone('America/Bogota')
fecha_y_hora_bogota = fecha_y_hora_actualrs.astimezone(fuso_horario)
fecha_y_hora_bogota_en_texto = fecha_y_hora_bogota.strftime('%d/%m/%Y %H:%M')

print(fecha_y_hora_bogota_en_texto)
```

Tenga en cuenta que ponemos el `timezone` como América / Bogota. ¿Pero qué pasa si queremos conocer otras posibilidades, como Ciudad de México, São Paulo, Buenos Aires?

Puede ver la lista de zonas horarias compatibles con `pytz` iterando sobre `pytz.all_timezones`:

```
import pytz

for tz in pytz.all_timezones:
    print(tz)
```

¡Trabajar con fechas para nosotros ya no será un problema!

Más Python y fechas

A lo largo de esta publicación, utilizamos varios códigos de formato de fecha, como el estándar %d/%m/%Y% H:%M. ¿Pero qué significa?

Estos códigos están definidos por la documentación del strftime (3). Los utilizados en nuestros ejemplos son:

- **%d**: el día del mes representado por un número decimal (del 01 al 31)
- **%m**: el mes representado por un número decimal (del 01 al 12)
- **%Y**: el año representado por un número decimal que incluye el siglo
- **%H** - La hora representada por un número decimal usando un reloj de 24 horas (de 00 a 23)
- **%M**: el minuto representado por un número decimal (de 00 a 59)

Conclusión

Comenzamos la publicación con la necesidad de manipular las fechas con Python y vimos cómo hacerlo usando el tipo `date`. Aprendimos cómo agregar nuestra fecha a una hora usando la clase `datetime`, y cómo formatear esta información en un string que nos sea fácil de leer.

Terminamos teniendo un problema con la zona horaria y vimos cómo resolverlo con la clase `timezone` en Python 3. También vimos una manera más simple de resolver este problema con la biblioteca `pytz`, creada por la comunidad Python.

¿Qué tal aprender más sobre **Python** y sus diversos recursos? Entonces, ¡Mira nuestros cursos de **Python para Data Science** aquí en Alura!

ARTÍCULOS DE TECNOLOGÍA > DATA SCIENCE

CURSOS

Cursos de Programación
Lógica de Programación | Java

Cursos de Front End
HTML y CSS | JavaScript | React

Cursos de Data Science
Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística

Cursos de DevOps
Docker | Linux

Cursos de Innovación y Gestión
Productividad y Calidad de Vida | Transformación Ágil | Marketing Analytics | Liderazgo y Gestión de Equipos | Startups y Emprendimiento